Arch Computat Methods Eng (2015) 22:557–576
DOI 10.1007/s11831-014-9126-8

ORIGINAL PAPER

Alya: Computational Solid Mechanics for Supercomputers

E. Casoni · A. Jérusalem · C. Samaniego · B. Eguzkitza · P. Lafortune · D. D. Tjahjanto · X. Sáez · G. Houzeaux · M. Vázquez

Received: 17 July 2014 / Accepted: 22 July 2014 / Published online: 13 August 2014

Abstract While solid mechanics codes are now conventional tools both in industry and research, the increasingly more exigent requirements of both sectors are fuelling the need for more computational power and more advanced algorithms. For obvious reasons, commercial codes are lagging behind academic codes often dedicated either to the implementation of one new technique, or the upscaling of current conventional codes to tackle massively large scale computational problems. Only in a few cases, both approaches have been followed simultaneously. In this article, a solid mechanics simulation strategy for parallel supercomputers based on a hybrid approach is presented. Hybrid parallelization exploits the thread-level parallelism of multicore architectures, combining MPI tasks with OpenMP threads. This paper describes the proposed strategy, programmed in Alya, a parallel multi-physics code. Hybrid parallelization is specially well suited for the current trend of supercomputers, namely large clusters of multicores. The strategy is assessed through transient non-linear solid mechanics problems, both for explicit and implicit schemes, running on thousands of cores. In order to demonstrate the flexibility of the proposed strategy under advance algorithmic evolution of computational mechanics, a non-local parallel overset meshes method (Chimera-like) is implemented and the conservation of the scalability is demonstrated.

Keywords Computational mechanics · Finite element method · Parallel computing · Chimera

E. Casoni (✉) · C. Samaniego · B. Eguzkitza · X. Sáez · G. Houzeaux · M. Vázquez
Barcelona Supercomputing Center (BSC-CNS), Edificio NEXUS I, Campus Nord UPC Gran Capitán 2–4, 08034 Barcelona, Spain
e-mail: eva.casoni@bsc.es

M. Vázquez
Artificial Intelligence Research Institute CSIC (IIIA-CSIC), Campus de la UAB, 08193 Bellaterra, Spain
e-mail: mariano.vazquez@bsc.es

A. Jérusalem · D. D. Tjahjanto
IMDEA Materials Institute, Tecnogetafe, C/ Eric Kandel 2, 28906 Getafe, Spain

D. D. Tjahjanto
Department of Solid Mechanics, KTH Royal Institute of Technology, Teknikringen 8D, 10044 Stockholm, Sweden

P. Lafortune
Idra Simulation S.C.P., 08004 Barcelona, Spain

A. Jérusalem
Department of Engineering Science, University of Oxford, Parks Road, Oxford OX1 3PJ, UK
e-mail: antoine.jerusalem@eng.ox.ac.uk

1 Introduction

With the increasing need for more advanced modeling techniques involving non-local approaches or multiphysics interactions, finite element (FE) solid mechanics codes requirements have traditionally slowed down the parallel efforts aimed at increasing the computational scalability of such codes. One of the direct consequences of this is that most commercial codes cannot efficiently scale on parallel computers when more than hundreds of cores are used [5,24]. Academic FE codes, on the other hand, have often relied on the need to develop one unique technique of interest, potentially followed by a secondary development phase aimed at scaling it up because of the prohibitive cost of the technique. As a direct consequence, researchers have often focused on the scalability of one technique independently of the other ones; see for example the Arbitrary Lagrangian-Eulerian (ALE) methods [51], Discontinuous Galerkin (DG) methods [21,71], fluid–structure interaction (FSI) [27,51,67] or even expensive constitutive models (and their also expensive meshing requirements) such as crystal plasticity [70]. It is also noticeable that the range of fields of application for these references is extremely wide, ranging from bio-medical, military, seismic, or fracture mechanics to polycrystalline texture analysis.

1.1 Fluid Mechanics Versus Solid Mechanics

Fluid mechanics computational codes have generally been exploiting advanced techniques conjointly with parallel efficiency (see for instance OpenFOAM [9], Alya [1] or CODE_SATURNE [4]). This can be explained by the fact that fluid problems have traditionally required larger meshes than Solid Mechanics problems but also by the fact that model requirements for solids may hinder the parallelization itself, for instance, in fracture mechanics. In that sense, the development of parallel Solid Mechanics codes requires a more complex structure in order, to not only solve the partial differential equation (PDE) involved, but also to add the complex features typical of computational Solid Mechanics: e.g., non-linear behavior of materials, non-locality of the model in fracture mechanics or continuum damage theory, non-linearity (and unphysical) forces in contact mechanics, multi-scale model needed in fracture and cracks, etc.

1.2 FE Solid Mechanics Codes

Designing computational models of complex material deformation mechanisms such as fracture, phase change or impact requires a combination of new numerical methods and improved computing power. Beyond the advanced numerical techniques, an increased computational power is needed to resolve the many length and time scales of these problems. Most of the modern FE codes can parallelize over distributed memory clusters to allow for solution on larger meshes or to reduce computation time. Sandia Labs has developed the SIERRA framework [77], which provides a foundation for several computational mechanics codes, for instance the highly scalable software Salinas [23] or the open source code CODE ASTER [3]. Other open source software such as Gmsh [36] or FEniCS [61] also offer PETSc [19] or Epetra [45]. Recently, however, shared memory multicore technology such as graphics processing units (GPUs) [55] have gained prominence due to their arithmetic ability and high power efficiency, such as in FEAST software package [41].

Large scale Solid Mechanics FE computation became a field of research on its own as soon as in the early 80's [37]. However, if preconditioner optimization studies for large scale computing were tackled shortly after [49], and barring a few exceptions (e.g., Ref. [35] focusing on XFEM), they have almost always been focused on conventional FE codes (and often for one unique application) without consideration of all the previously listed evolutions [17,33,60,79]. In the particular case of FE for Solid Mechanics, many computer codes were written to solve one specific application and are not flexible enough to adapt the newest schemes and technological improvements. These limitations of code structure make it difficult to integrate the latests research that is required to solve ever larger and more complicated problems. Instead, the rapid advances on parallel computation require the code to be developed from the beginning for *any* large scale application, and then enhanced with advanced techniques following the same large scale framework imposed by the code structure.

1.3 System of Equations

The cornerstone of any FE method involves an assembly and a solution of a linear or non-linear system of equations. In an implicit scheme, this solution process requires solving a liner system of equations, which usually has dimension of millions and even billions of degrees of freedom. The solution of large sparse linear systems of equations is the main concern in any parallel code [43]. The algorithms to solve these systems are basically grouped into two categories: direct methods and iterative methods. For decades, research has mainly focused on taking advantage of sparsity to design direct methods that can be economical. However, a substantial portion of the total computational work and storage required to solve stiff initial value problems is devoted to solve these linear algebraic systems, particularly if the system is large (see for instance the study in Ref. [38]). However, over the last decade, several efficient iterative methods have been developed to solve large sparse (non-symmetric) systems of linear algebraic equations that involve a large number of variables (sometimes of the order of millions) [72]. In these cases, direct methods would be prohibitively expensive even with the best available computing power and iterative methods appear as the only rational alternative.

1.4 Algebraic Solvers

An algebraic system $\mathbf{Ax} = \mathbf{b}$ can be approached by two categories of methods. Krylov subspace methods tackle the problem directly by solving directly for $\mathbf{x}$. Some examples are the conjugate gradient (CG) method for the symmetric cases, generalized minimal residual (GMRes) and the biconjugate gradient (BiCGSTAB) method for the non-symmetric cases [72]. The other family of methods are the primal and dual Schur complement methods, which solve for the interface unknowns, the restriction of $\mathbf{x}$ on the subdomain interfaces and its dual (e.g. the FETI method [32,73]), respectively. For this last class of methods, Krylov solvers can then be used to solve the interface problem. In both approaches, a preconditioner is also required to enhance the convergence. Examples of classical preconditioners are the Gauss-Seidel, block diagonal, Deflated [62] or Multigrid [13,57,66]. The preconditioner can also be based on domain decomposition methods (DD) as well. Some examples are: Schwarz, Restricted Additive Schwarz (RAS), Block LU, Imcomplete Block LU, where the support of the preconditioners coincide with the subdomains of the mesh partition.

Some definitions are now introduced to assess the efficiency of a solver plus a preconditioner in a parallel environment. Let np and dof be the number of parallel processes and the number of degrees of freedom of the algebraic system, respectively. If $t_{np}(dof)$ is the time to achieve the convergence criterion of the solver, the efficiency of the *strong scalability* is defined as as

$$\text{Efficiency of strong scalability} = \frac{t_1(dof)}{np \times t_{np}(dof)}.$$

Thus, a solver with a unit efficiency enables one to achieve convergence np faster by using np processes, i.e. $t_{np}(dof) = t_1(dof)/np$. Krylov solvers with classical preconditioners can be made strongly scalable, as long as dof is sufficient large on each process, in order to keep a high *work versus communication* ratio. However, if one wishes to increase the global number of degrees of freedom by refining the mesh, the strong scalability does not provide any information about the computing time. The observation of *weak scalability* means that the amount of time budged does not increase if one maintains a fixed number of dof per processor. In this case perfect efficiency is achieved if the run time stays constant while the workload (dof) is increased in direct proportion to the number of processors:

$$\text{Efficiency of weak scalability} = \frac{t_1(dof)}{t_{np}(np \times dof)}.$$

This criterion is useful to estimate the CPU time while refining the mesh and answer the following question: if one refines the mesh by a given factor, how many more CPUs should one use to converge at the same computing time. Even though it is relatively easy to maintain the number of iterations of the solver independently of dof, it is much harder to maintain the computing time. This is due to the fact that complex preconditioners must be used, in addition to some coarse space solver to propagate information across the CPUs. These coarse space solvers are usually the bottleneck of the weak scaling [18,39].

It must be emphasized that some solvers present the same convergence independently of np. This is the case for example of Krylov methods together with a diagonal preconditioner. Weak scalable solvers do not, as the mesh partition is the base for the construction of the local solvers. Therefore, they do not enable reproducibilty. This means that if a simulation is carried out on 1,024 CPUs on one day, different convergence or even results may be obtained on another day, using 256 CPUs. This is not a trivial remark as reproducibility is a very relevant aspect in the industry.

For some applications leading the very stiff systems, direct solvers could even be required. Direct methods appear as feasible solution for solving linear systems that are ill-conditioned. Actually, they are commonly used in implicit FE codes for structural dynamic problems. However, these methods were initially discarded due to their high storage memory requirements. In the past the limitations of CPU and memory requirements have prevented the use of these methods but nowadays, with the increasing advances in power computing new trends on direct solvers are being investigated. These are either multifrontal [16] and supernodal techniques [34]. Multifrontal techniques are basically the multifrontal massively parallel solver (MUMPS) [28] and the Watson Sparse Matrix package (WSMP)[40]. SuperLUDIST [58] is a MPI parallel version of SuperLU family of solvers for unsymmetric systems based on supernodal right LU factorization. A list of the available software for solving sparse linear systems via direct methods is presented in Ref. [12]. Performance results of these iterative solver are compared against a direct solver routine of the commercial Harwell Software Library [53].

Many studies on the behavior of iterative solvers in classical Solid Mechanics problems have been carried out. For instance, a comparison was made between preconditioned conjugate gradients (PCG) and Multigrid methods (MG), applying them to a series of test problems of plane elasticity [52]. Different iterative solvers for large scale linear algebraic systems for 3D elasticity are compared in Ref. [30]. Solving nonlinear Solid Mechanics problems with a Newton-type method could be problematic if the determinations, storage or solution cost associated with the Jacobian is high. The Jacobian Free Newton–Krylov methods [54] initially developed for CFD problems has been applied to nonlinear computational Solid Mechanics in Ref. [42]. Multigrid methods, either as iterative methods or as a preconditioners, have been studied in many application areas: for the Helmholtz equation when analyzing frequency responses of an structure [14], for plasticity problems [85] or recently in fracture mechanics with the XFEM method [83]. In contrast, direct solvers are not as popular as iterative solvers. MUMPS has been used for simulations of linear elasticity coupled with acoustics in [69]. SuperLU was used to solve FSI problems with large- displacements in Ref. [44]. As a conclusion, despite some recent efforts, more focused research is needed to analyze the solvers behavior and requirements in complex structures or in the recent techniques for the specific features of Solid Mechanics problems, such as fracture mechanics.

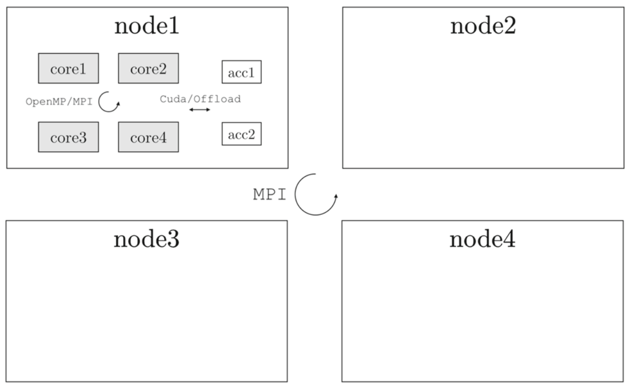

Fig. 1 General architecture of a supercomputer: nodes, cores and accelerators, and associated common programming languages

1.5 Parallel Context

High performance computational mechanics codes exploit one or more levels of parallelism offered by modern supercomputers. A general supercomputer architecture is depicted in Fig. 1, together with some common programming languages. At the coarse level, the architecture consists of nodes, in a distributed memory environment. At the medium level, each node is composed of a given number of cores, in a shared memory environment. These nodes can, in turn, use the accelerators (e.g. GPUs or Xeon Phi.) as a fine level of parallelism [68].

MPI is the most commonly used Message Passing Library on distributed memory architectures while OpenMP [10] is the preferred one for shared memory architectures, i.e. parallelizing the computation at the core level. Accelerators can be programmed using programming languages like OpenCL [8], OpenACC [7] or CUDA [63]. Very few unstructured mesh computational mechanics codes have been ported to accelerators [2,26,46]. There are three main reasons for this: the huge cost of porting large codes; the difficulty in mapping codes with irregular memory access to the vector architecture (SIMD) of accelerators; the lack of a standard language [25]. One standard candidate is the new version of OpenMP, namely OpenMP4 (see new advances in [10]). On the other hand, Intel solution to accelerators, the Xeon Phi, presents very appealing features for unstructured meshes due to its very low programming effort together with its computing power. See Ref. [82] for more complete assessment Solid Mechanics problems.

1.6 Libraries

Many different software packages are available to solve a linear system of equations. Most modern solid mechanic codes include an abstract interface to a LinearSolver class that operates on a sparse Matrix and a Vector for the right hand side. Inherited from this LinearSolver base class are different solvers that can interface to packages. In the top of these packages are PETSc [20], which supports also MPI and GPU architectures, and Hypre [31], which provides also Multigrid preconditioning in a parallel MPI environment. Moreover, MUMPS methods are also included in a direct solver library developed in Ref. [16], which is supported by MPI. The WSMP software package WSMP [40] is also developed using an hybrid implementation with MPI and P-threads, among others. Abstracting the solver interface in this manner allows the application writer to choose the most effective solver for the particular problem.

1.7 Alya

Alya [1] has been conceived as a Computational Mechanics code developed at Barcelona Supercomputing Center (BSC–CNS) and aimed at solving PDEs in unstructured meshes. Alya exploits the similarities of the PDE-governed problems to solve with high parallel performance compressible and incompressible flows, thermal flows, excitable media or quantum mechanics for transient molecular dynamics [29,47,48,81], while running in thousands of cores. Parallelization is hidden behind a common solver that assembles matrices and residuals and carries out the solution scheme. The scalability of the code thus exclusively depends on one unique set of parallel communication subroutines independently of the physics of the problem. Additionally, Alya is specially well-suited for the parallel solution of coupled multi-physics problems.

In the present work, the Solid Mechanics module of Alya is introduced. Using the large scale PDE solver capability of Alya's kernel, the solid module was implemented so that any future development of more complex FE techniques should conserve the high scalability of the code. These developments have been carried out along two parallel strategies: first, the MPI implementation at the node level, where the parallelization is based on a sub-structuring technique and uses a Master–Worker strategy [48]; and secondly, the OpenMP implementation used to treat many-core processors. The overall code is thus hybrid MPI/OpenMP. Non-symmetric problems are solved using GMRes or Bi-CGStab schemes and symmetric ones, solved using CG methods [72]. There are several available preconditioners: diagonal, Deflated, RAS, blockLU, etc.

The Alya framework is specially well-suited to solve coupled problems. A Chimera overset meshes strategy is a particular way of coupling problems: two or more problems are solved in different meshes which are connected using their respective geometrical information. This connection scheme results in the fusion of all the problems in one single matrix. The problem can then be solved using the same parallelization scheme proposed earlier [29]. This scheme was thus implemented in Alya as an illustration of its large scale parallelization strategy where the code structure naturally lends itself to complex non-local approaches without loss of scalability performance.

The article is structured as follows. Section 2 introduces the governing equations and briefly describes the numerical method. Section 3 describes the structure of Alya, with special attention on the parallel strategy and the solver in Sect. 4. The parallelization and hybrid strategies are formally presented in Sect. 5 and the optimal scalability of the code is also shown. Section 6 presents some problems of applications where the Alya modules have been used with good performance. The paper ends with some conclusions and future lines.

2 Numerical Scheme

The computational solid mechanics problem is solved using a standard Galerkin method for a large deformation framework using a generalized Newmark time integration scheme. This framework, developed in a total Lagrangian formulation, is only briefly summarized below; for more details, see Ref. [59].

2.1 Standard Galerkin Governing Equations

Let $\boldsymbol{\varphi} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ be the function that maps a material point of an undeformed body $\boldsymbol{X} \in B_0$ in the reference configuration to its point $\boldsymbol{x} = \boldsymbol{\varphi}(\boldsymbol{X}) \in B$ in the current (or deformed) configuration. The deformation gradient tensor $\boldsymbol{F}$ is defined as $\boldsymbol{F} := \nabla_0 \boldsymbol{x}$, where ∇_0 is the gradient operator with respect to the reference configuration. In cartesian basis, the components of $\boldsymbol{F}$ are given by $F_{iJ} = \partial x_i / \partial X_J$. Index i in vector $\boldsymbol{x}$ and J in vector $\boldsymbol{X}$ are written in lower and upper case to indicate that the component is referred to in the current and reference configurations, respectively. Since $\boldsymbol{x}(\boldsymbol{X}) = \boldsymbol{X} + \boldsymbol{u}(\boldsymbol{X})$, where $\boldsymbol{u}$ is the displacement vector, the deformation gradient can be given by $\boldsymbol{F} = \boldsymbol{I} + \nabla_0 \boldsymbol{u}$, with $\nabla_0 \boldsymbol{u}$ as the displacement gradient.

The equation of balance of momentum with respect to the reference configuration can be written as

$$\text{Div}\, \boldsymbol{P} + \boldsymbol{b}_0 = \rho_0 \ddot{\boldsymbol{u}} \ , \quad \forall \boldsymbol{X} \in B_0 \ , \tag{1}$$

where ρ_0 is the mass density (with respect to the reference volume) and Div is the divergence operator with respect to the reference configuration, with Div $\boldsymbol{P} = \nabla_0 \cdot \boldsymbol{P}$. Tensor $\boldsymbol{P}$ and vector $\boldsymbol{b}_0$ stand for the first Piola–Kirchhoff stress and the distributed body force on the undeformed body, respectively. Prescribed displacements and tractions are applied at reference boundary $\Gamma_0 = \Gamma_{d0} \cup \Gamma_{n0}$, where Γ_{d0} and Γ_{n0} correspond to the Dirichlet and Neumann boundary conditions, respectively:

$$\boldsymbol{u} = \bar{\boldsymbol{u}} \ , \quad \forall \boldsymbol{X} \in \Gamma_{d0} \ , \tag{2}$$

$$\boldsymbol{P} \cdot \boldsymbol{N}_0 = \bar{\boldsymbol{t}}_0 \ , \quad \forall \boldsymbol{X} \in \Gamma_{n0} . \tag{3}$$

where $\boldsymbol{N}_0$ is the normal to the boundary in the reference configuration.

As usual in Finite elements methods [86], the weak form of balance of the momentum (1) can be formulated for any arbitrary admissible virtual displacement $\boldsymbol{w}$, such that,

$$\int_{B_0} \boldsymbol{P} \cdot \nabla \boldsymbol{w} \, \mathrm{d}V + \int_{B_0} \rho_0 \ddot{\boldsymbol{u}} \cdot \boldsymbol{w} \, \mathrm{d}V = \int_{B_0} \boldsymbol{b}_0 \cdot \boldsymbol{w} \, \mathrm{d}V + \int_{\Gamma_{n0}} \boldsymbol{w} \cdot \boldsymbol{t}_0 \mathrm{d}\Gamma \tag{4}$$

For a FE approximation $\Omega_0 = \bigcup_e \Omega_0^e$ of the undeformed continuum body B_0, let u_h be a polynomial approximation of degree k to the actual displacement $\boldsymbol{u}$

$$u_h(\boldsymbol{X}) = \mathbf{N}(\boldsymbol{X}) \, \mathbf{u}_h , \tag{5}$$

where $\mathbf{u}_h$ denotes time-dependent nodal displacement and $\mathbf{N}$ the three-dimensional matrix of shape functions.

Then, the discrete form of (4) consists in solving for the array of nodal displacement $\mathbf{u}_h \in \mathbb{R}^3$

$$\mathbf{M}\ddot{\mathbf{u}}_h + \mathbf{f}_{int} = \mathbf{f}_{ext} \tag{6}$$

where $\mathbf{M}$, $\mathbf{f}_{int}(\mathbf{u}_h)$, and $\mathbf{f}_{ext}$ are, respectively, the mass matrix and the vector of internal and external forces vector. For more references, see [59].

The generalized Newmark formulation used here for Eq. (6) can be written as [59]

$$\mathbf{M}\ddot{\mathbf{u}}_h^{n+1} + \mathbf{f}_{int}^{n+1} = \mathbf{f}_{ext}^{n+1} \ , \tag{7}$$

$$\mathbf{u}_h^{n+1} = \mathbf{u}_h^n + \Delta t \dot{\mathbf{u}}_h^n + (\Delta t)^2 \left[\left(\frac{1}{2} - \beta \right) \ddot{\mathbf{u}}^n + \beta \ddot{\mathbf{u}}_h^{n+1} \right] , \tag{8}$$

$$\dot{\mathbf{u}}_h^{n+1} = \dot{\mathbf{u}}_h^n + \Delta t \left[(1-\gamma) \, \ddot{\mathbf{u}}_h^n + \gamma \ddot{\mathbf{u}}_h^{n+1} \right] , \tag{9}$$

where superscripts "n" and "$n+1$" indicate that the variables are evaluated at time t^n and t^{n+1}, respectively. Parameters β and γ set the characterictics of the Newmark scheme. Parameter $\beta = 0$ leads to an explicit Newmark scheme, whereas for values $0 < \beta \leq 0.5$ leads to an implicit scheme. In the latter case, the set of equations (7)–(9) are solved for the unknown displacement $\mathbf{u}_h^{n+1}$, velocity $\dot{\mathbf{u}}_h^{n+1}$, and acceleration $\ddot{\mathbf{u}}_h^{n+1}$ using the iterative Newton-Raphson algorithm.

3 Numerical Implementation

The Alya system is a computational mechanics code developed with two main motivations. First, it was designed to

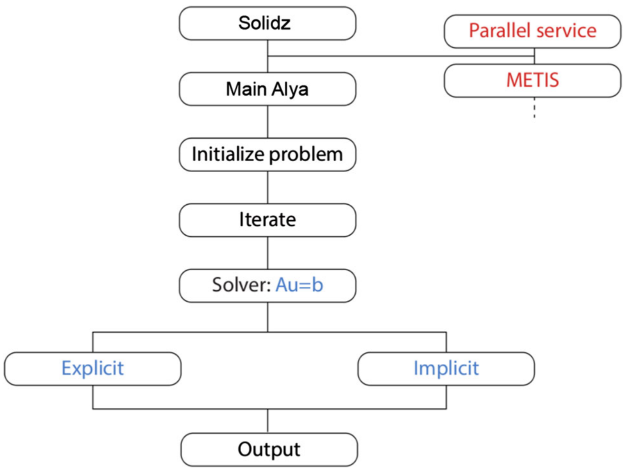

Fig. 2 *solidz* module structure in Alya; *kernel* elements are in *blue* and *services* in *red*. (Color figure online)

run with high efficiency standards in large scale supercomputing facilities. Secondly, various physical problem should be allowed to be solved individually or in a coupled manner, while conserving exceptional scalability and retaining their individual efficiency.

Alya's architecture is modular, grouping the different tasks into *kernel*, *modules* and *services*. The *kernel*, the core of Alya, contains all the facilities required to solve any set of discretized PDEs (e.g., the solver, the I/O, the coupling, the elements database, the geometrical information, etc.).

The physical description of a given problem is provided by its corresponding *module* (e.g., the discretized terms of the PDE, the meaning of the boundary and initial conditions, etc.). Other Alya modules than the one presented here include incompressible or compressible flow, thermal transport, or N-body problem, among others. Finally, the *services* contain the toolboxes providing several independent procedures to be called by *modules* and *kernel* (e.g., parallelization or optimization).

3.1 Alya's Solid Mechanics Module

A large database of element types and integration schemes of different orders is available from previous work in the other modules. Well known constitutive equations for large deformation elasticity constitutive models, such as the neo-Hookean or specific hyperelastic models, were also implemented.

Figure 2 shows a flowchart of the *solidz* module. All the geometrical and physical data of the problem are introduced as input files. Once read, Alya initializes the computation within the *solidz* module, either in serial or parallel mode. The parallel service must be specified in the input files.

3.2 Mesh Convergence

The following test aims at studying the mesh convergence of the Alya-*solidz* module. To this end, a target solution $\boldsymbol{u}^{(e)}$ with a given degree of regularity is used. By using a stationary manufactured solution belonging to the finite element space (for linear and bilinear elements), the numerical scheme should provide an exact nodalwise solution, i.e., $\boldsymbol{u} = \boldsymbol{u}^{(e)}$, thus a) confirming that the equation is well coded and b) allowing for the proposed mesh convergence study. The analysis below uses indicial notation and the following equation, obtained from Eq. (1), is solved:

$$\rho_0 \ddot{u}_i - \frac{\partial P_{iJ}}{\partial X_J} = -\frac{\partial P_{iJ}^{(e)}}{\partial X_J}, \tag{10}$$

where $P_{ij}^{(e)} = P_{ij}(\boldsymbol{u}^{(e)})$. Assuming large deformations and using a linear elastic constitutive law, i.e., the last term of Equation (10) is given by:

$$\frac{\partial P_{iJ}^{(e)}}{\partial X_J} = C_{KJML}\left[\frac{F_{iK}^{(e)}}{2}\left(\frac{\partial^2 u_n^{(e)}}{\partial X_J \partial X_M}F_{nL}^{(e)} + \frac{\partial^2 u_n^{(e)}}{\partial X_J \partial X_L}F_{nM}^{(e)}\right) + \frac{\partial^2 u_i^{(e)}}{\partial X_K \partial X_J}E_{ML}^{(e)}\right] \tag{11}$$

where C_{KJML} is the fourth order elasticity tensor and F_{nL} (or F_{iK} and F_{nM}) and E_{ML} are the deformation gradient and the Green-Lagrange strain tensors, defined as:

$$\begin{aligned} F_{mL}^{(e)} &= \frac{\partial u_m^{(e)}}{\partial X_L} + \delta_{mL} \\ E_{ML}^{(e)} &= F_{nM}^{(e)} F_{nL}^{(e)} - \delta_{ML} \end{aligned} \tag{12}$$

The 2D manufactured solution considered here is arbitrarily chosen to be:

$$\begin{cases} u_1^{(e)} = X_1^3 X_2^4 \\ u_2^{(e)} = X_1^3 X_2^3 \end{cases} \tag{13}$$

and is solved on a unit square. Figure 3 shows the resulting mesh convergence results computed on a regular mesh of $Q1$ (quadrilateral) elements. The L^2-norm of the error and of the error of the gradient, $||\epsilon(\boldsymbol{u})||_{L^2}$, and $||\epsilon(\nabla \boldsymbol{u}||_{L^2}$, respectively, are shown and they are computed as follows:

$$||\epsilon(\boldsymbol{u})||_{L^2} = \sqrt{\int_\Omega (\boldsymbol{u}_h - \boldsymbol{u}^{(e)})^2 d\Omega} \Big/ \sqrt{\int_\Omega \boldsymbol{u}^{(e)2} d\Omega},$$

$$||\epsilon(\boldsymbol{u})||_{H^1} = \sqrt{\int_\Omega (\nabla \boldsymbol{u}_h - \nabla \boldsymbol{u}^{(e)})^2 d\Omega} \Big/ \sqrt{\int_\Omega (\nabla \boldsymbol{u})^{(e)2} d\Omega}.$$

The results successfully confirm a first order convergence in the derivative and second order in the L^2-norm of the displacement with respect to h for the *solidz* module [50].

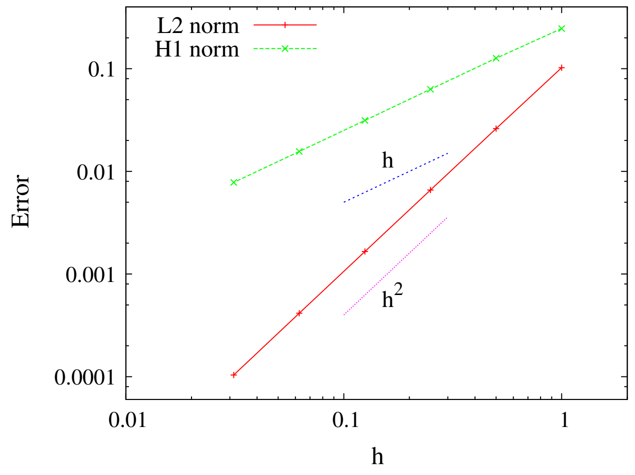

Fig. 3 Mesh Convergence study; L^2-error of the displacement and of the gradient of the displacement

4 MPI Parallelization

4.1 Parallel Context

The parallelization for distributed memory supercomputers is based on a domain decomposition technique, using a Master–Workers paradigm and MPI as the message-passing library. In the core of the parallelization outer layer lies the automatic graph partition tool, METIS [6]. First, the master reads the mesh and performs the partition of the mesh into submeshes, or subdomains, using METIS. Each of these subdomains is called a Worker. The Master distributes each partition among the Workers that carry out the solution computational work. METIS mesh partition is done by maximizing load balance and minimizing communication (see Ref. [48] for more details). As a second step, the Workers build the local elements matrices and local right-hand sides, and are in charge of solving the resulting system solution in parallel. In the assembling tasks, no communication is needed between the Workers, and the scalability only depends on the load balancing. In the iterative solvers, the scalability depends on the size of the interfaces and on the communication scheduling.

Depending on the constitutive model, the resulting equations can be symmetric or non-symmetric. In the implicit case, the basic iterative solvers are GMRes and CG [72]. During the execution of these iterative solvers, two main types of asynchronous communications are required:

- Point-to-point communications via `MPI_ISend` and `MPI_IRecv`, which are used when sparse matrix-vector products are calculated.
- Collective communications via `MPI_AllReduce`, which are used to compute residual norms and scalar products.

In the current implementation of Alya, the solution obtained in parallel is, up to round-off errors, the same as the sequential one all the way through the computation. This is because the mesh partition is only used for distributing work without in any way altering the actual sequential algorithm. This would not be the case if one would consider more complex solvers, like primal or dual Schur complement solvers [64], or more complex preconditioners, like linelet [74] or block LU [65]. Since the explicit framework is relatively straightforward to implement, in the following we only focus our attention on the implicit framework.

The numerical solution of a PDE (and consequently the solid mechanics equations) consists mainly of two steps: first, the construction of the matrix $\mathbf{A}$ and right-hand side (RHS) $\mathbf{b}$ of the algebraic system $\mathbf{Ax} = \mathbf{b}$; second, the solution of this system using an iterative solver. As far as the matrix and RHS assemblies are concerned, only part of the matrix is assembled for the interface nodes. For the iterative solvers, the basic operations are the matrix-vector and the dot products. Let us consider these two operations for a simple one-dimensional case illustrated in Fig. 4.

In the context of FE, the coefficients of the matrix come from element computations (see Sect. 2.1). The contribution of node 3 comes from subdomain 1 and 2, namely A^1_{33} and A^2_{33}, respectively. Since

$$y_3 = A_{32}x_2 + A_{33}x_3 + A_{34}x_4$$

and by rewritting it as

$$\begin{aligned} y_3 &= (A_{32}x_2 + A^{(1)}_{33}x_3) + (A^{(2)}_{33}x_3 + A_{34}x_4) \\ &= y_3^{(1)} + y_3^{(2)} \end{aligned}$$

the parallelization only consists of a residual exchange.

Therefore, the idea is to use the distributive law of the multiplication to carry out the matrix-vector product in parallel. Let us introduce two functions, that will be described formally in the next subsection as **PAREXC** and **PARASM**. In the previous 1D example, the asynchronous matrix-vector product can be carried out in parallel as follows:

- Perform local matrix-vector product of boundary nodes (node 3);
- **PAREXC**: Exchange the results on the interface $y_3^{(1)}$ and $y_3^{(2)}$ using non-blocking MPI functions;
- Perform local matrix-vector product of interior nodes (rows 1 and 2 for subdomain 1, 4 and 5 for subdomain 2);
- Synchronization MPI_Waitall;
- **PARASM**: Assemble (sum) the local contributions of each subdomain: $y_3 = y_3^{(1)} + y_3^{(2)}$.

Regarding the dot product, i.e., $x \cdot y$ we only need to introduce the concept of *own* interface node. In the current

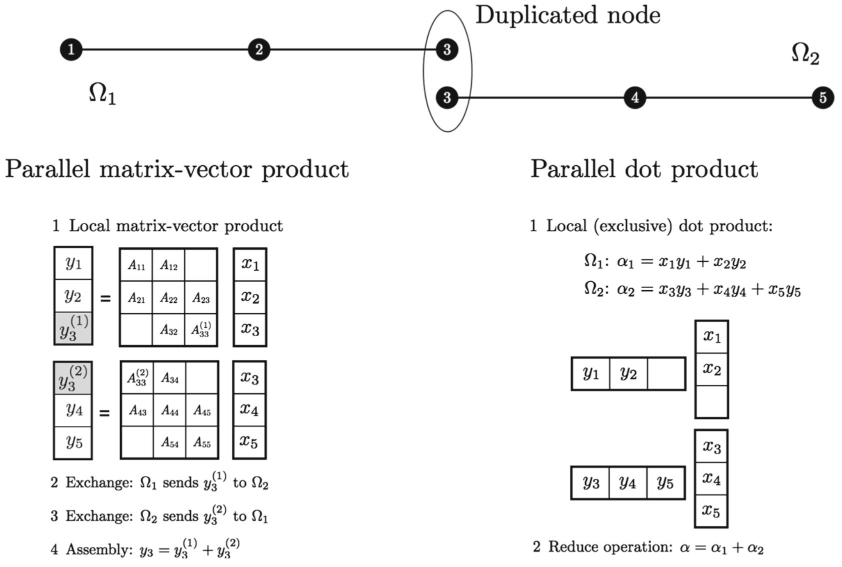
Duplicated node
Ω_1
Ω_2
Parallel matrix-vector product
Parallel dot product
1 Local matrix-vector product
1 Local (exclusive) dot product:
Ω_1: $\alpha_1 = x_1y_1 + x_2y_2$
Ω_2: $\alpha_2 = x_3y_3 + x_4y_4 + x_5y_5$
2 Exchange: Ω_1 sends $y_3^{(1)}$ to Ω_2
3 Exchange: Ω_2 sends $y_3^{(2)}$ to Ω_1
4 Assembly: $y_3 = y_3^{(1)} + y_3^{(2)}$
2 Reduce operation: $\alpha = \alpha_1 + \alpha_2$

Fig. 4 Iterative solver basic operations: matrix-vector product and dot product

implementation all vectors are assembled on the interface, which implies:

$$x_3^{(1)} = x_3^{(2)} = x_3 \text{ and } y_3^{(1)} = y_3^{(2)} = y_3$$

Then, if both subdomains compute their entire local dot product, then the sum of the two contributions will account for x_3y_3 twice, hence

$$\alpha = x_1y_1 + x_2y_2 + 2\,x_3y_3 + x_4y_4 + x_5y_5,$$

which provides a wrong result. In order to account only once for the interface value, all the interface nodes are splitted into *own* interface nodes and *oth* (for other) interface nodes. That is, an interface node has the *own* status only in one subdomain although it can be shared by others, but with status *oth*. The local dot product thus only involves interior nodes and interface nodes of *own* status.

In the next section we define formally the concept of interior and interface nodes and introduce some notations and operators, as well as the functions **PAREXC** and **PARASM** doing the parallel matrix-vector and dot products. The term boundary will refer to the interfaces between subdomains.

4.2 Formal set definitions

Before introducing the parallel operators, it is first necessary to define the basic sets, categorizing the nodes and elements of each subdomain, as well as the relations between the subdomains. Some of the definitions are illustrated in Fig. 5.

Consider a computational domain $\Omega \in \mathbb{R}$. Let $\mathcal{N} = \{n_1, n_2, \dots, n_N\}$ and $\mathcal{E} = \{e_1, e_2, \dots, e_E\}$ be the sets of all nodes and elements, respectively, of the FE mesh used to discretize the computational domain. Here, N and E denote

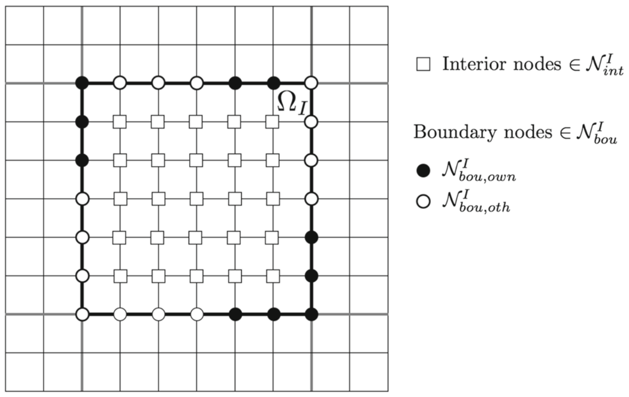
Ω_I
□ Interior nodes $\in \mathcal{N}^I_{int}$
Boundary nodes $\in \mathcal{N}^I_{bou}$
● $\mathcal{N}^I_{bou,own}$
○ $\mathcal{N}^I_{bou,oth}$

Fig. 5 Node sets

the total number of nodes and elements of the computational mesh.

Any element $e \in \mathcal{E}$ can be defined as a tuple of nodes $e = (n_1^e, n_2^e, \dots, n_k^e)$ where k is the number of nodes per element.

Two sets are defined to describe the nodal connectivity with elements and nodes of the mesh, $\mathcal{L}_{ele}(n)$ and $\mathcal{L}_{nod}(n)$, respectively. For any arbitrary node $n \in \mathcal{N}$:

Definition Element connectivity of *n*. Let $\mathcal{L}_{ele}(n)$ denote the set of elements in $\mathcal{E}$ directly connected to the node n. Formally,

$$\mathcal{L}_{ele}(n) = \{e \in \mathcal{E} : n \in e\}. \tag{14}$$

Definition Node connectivity of *n*. Let $\mathcal{L}_{nod}(n)$ denote the set of nodes in $\mathcal{N}$ directly connected to n. Formally,

$$\mathcal{L}_{nod}(n) = \{m \in \mathcal{N} : \exists e \in \mathcal{L}_{ele}(n), m \in e\} \setminus \{n\} \tag{15}$$

The latter set represents the nodal connection of the mesh and the matrix graph as well. It contains the information required to construct the Compressed Sparse Row (CSR) format used to assemble the sparse matrix of the algebraic system.

We then consider a domain decomposition by elements (see the example of Sect. 4.1). In this work, the METIS [6] library is used. Let $\mathcal{N}^I$ and $\mathcal{E}^I$ denote the set of nodes and elements of any arbitrary subdomain Ω_I, respectively. The total number of nodes and elements of the mesh can be grouped by subdomains:

$$\mathcal{E} = \bigcup_{I=1}^{S} \mathcal{E}^I, \qquad \mathcal{N} = \bigcup_{I=1}^{S} \mathcal{N}^I,$$

where S is the number of subdomains. As the domain decomposition is performed by elements, the nodes can be shared between subdomains, namely boundary nodes, but the element subdomains are non-overlapping.

The interface created by the partition of the mesh divides the set of nodes of any arbitrary subdomain Ω_I into two disjoint sets. On the one hand, the set of interior nodes:

Definition (*Interior nodes of* $\boldsymbol{\Omega}_I$.) Let $\mathcal{N}^I_{int}$ denote the set of interior nodes of the subdomain Ω_I. Formally,

$$\mathcal{N}^I_{int} = \mathcal{N}^I \backslash \left(\bigcup_{J=1, J\neq I}^{S} \mathcal{N}^J \right).$$

These nodes do not belong to the boundary.

On the other hand, the set of boundary nodes:

Definition (*Boundary nodes of* $\boldsymbol{\Omega}_I$.) Let $\mathcal{N}^I_{bou}$ denote the set of boundary nodes of Ω_I. Formally,

$$\mathcal{N}^I_{bou} = \mathcal{N}^I \backslash \mathcal{N}^I_{int}$$

These nodes belong to the interface and are shared by different subdomains, including Ω_I.

In this new context of domain decomposition, let us define a subset of the set $\mathcal{L}_{nod}(n)$ related with any boundary node $n \in \mathcal{N}^I_{bou}$ as:

Definition (*Node connectivity of* $\boldsymbol{n}$ *belonging to* $\boldsymbol{\Omega}_I$.) Let $\mathcal{L}^I_{nod}(n)$ denote the set of nodes in $\mathcal{N}^I$ directly connected to n. Formally,

$$\mathcal{L}^I_{nod}(n) = \left\{ m \in \mathcal{N}^I : \exists e \in \mathcal{L}_{ele}(n), m \in e \right\} \backslash \{n\} \qquad (16)$$

The concept of adjacency between subdomains is defined as

Definition (*Adjacent subdomains*.) Two arbitrary subdomains I and J are adjacent when $\mathcal{N}^I \cap \mathcal{N}^J \neq \emptyset$.

In order to carry out the scalar product we also need to introduce the concept of *own* boundary and *oth* boundary, where *oth* is defined in Sect. 4.1. This is achieved by splitting the interfaces between the subdomains that share it, for example with METIS.

Definition (*Own and other's boundaries of* $\boldsymbol{\Omega}_I$.) Let use define $\mathcal{N}^I_{bou,own}$ and $\mathcal{N}^I_{bou,oth}$ such that

$$\mathcal{N}^I_{bou,own} \cap \mathcal{N}^I_{bou,oth} = \emptyset,$$
$$\mathcal{N}^I_{bou,own} \cup \mathcal{N}^I_{bou,oth} = \mathcal{N}^I_{bou},$$
$$\mathcal{N}^I_{bou,own} \cap \bigcup_{J=1, J\neq I}^{S} \mathcal{N}^J_{bou,own} = \emptyset.$$

4.3 Parallel Operators

We now define the parallel operator used in the algebraic solver to carry out the matrix-vector product and assembly contributions in an asynchronous way.

The node and element numbering is represented by a local index $index^I$, and an index used to exchange information between two neighboring subdomains I and J, $index^{I,J}_{bou}$.

For any arbitrary subdomain Ω_I,

Definition (*Local node numbering in*$\boldsymbol{\Omega}_I$.) Let the function

$$index^I : \mathcal{N}^I \rightarrow \{1, 2, 3, \ldots, |\mathcal{N}^I|\} \qquad (17)$$

be the local node numbering in Ω_I defined as $index^I(n) = i^I$ where $n \in \mathcal{N}^I$ and $i^I \in \mathbb{N}$. $|\mathcal{N}^I|$ is the cardinal number of the set.

For implementation purposes, the interior nodes are first numbered followed by the boundary nodes, as shown in Fig. 6. This ordering is useful to carry out not only the matrix-vector product in an efficient way but also the dot product.

Note that for two arbitrary adjacent subdomains Ω_I and Ω_J, $index^I(n)$ is not necessarily equal to $index^J(n)$ for any boundary node $n \in \mathcal{N}^I_{bou} \cap \mathcal{N}^J_{bou}$.

Definition (*Shared boundary node numbering in* $\boldsymbol{\Omega}_I$ and $\boldsymbol{\Omega}_J$.) Let the function

$$index^{I,J}_{bou} : \mathcal{N}^I_{bou} \cap \mathcal{N}^J_{bou} \rightarrow \{1, 2, 3, \ldots, |\mathcal{N}^I_{bou} \cap \mathcal{N}^J_{bou}|\} \qquad (18)$$

be the shared boundary node numbering for two arbitrary adjacent subdomains Ω_I and Ω_J defined as $index^{I,J}(n) = i \in \mathbb{N}$ where $n \in \mathcal{N}^I_{bou} \cap \mathcal{N}^J_{bou}$.

This index facilitates the data exchange between two subdomains and is constructed in such a way that both subdomains are able to order their shared boundary nodes in the

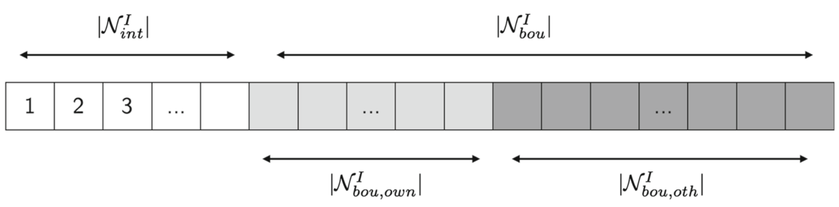

Fig. 6 Node numbering of subdomain Ω_I

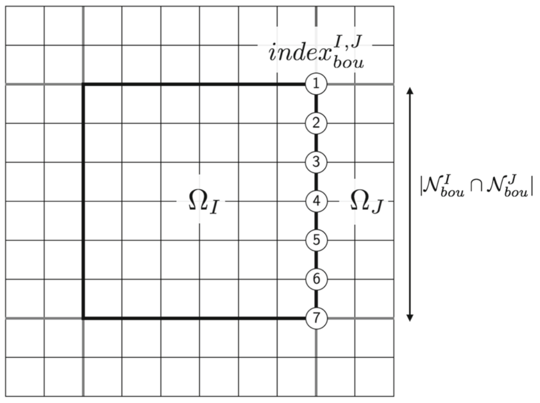

Fig. 7 Shared node numbering $index^{I,J}(n)$ of the interface nodes between Ω_I and Ω_J

same way, see Fig. 7. In Alya, this task is carried out by the master process, after the partition of the mesh.

Taking into account the previous defined functions, the implementation details of the operators **PAREXC** and **PARASM** from the point of view of any arbitrary subdomain Ω_I are provided in Algorithms 1 and 2. As the exchange of data between subdoamins is asynchronous, the operator is divided into two parts: the data exchange first, given by Algorithm 1, and the assembly operation, given by Algorithm 2.

Algorithm 1 The parallel operator **PAREXC** from Ω_I: data exchange

```
Input: a numeric array $data$ with length $\mathcal{N}^I$
Output: void
for each adjacent subdomain $J$ of $I$ do
    for each node $n \in \mathcal{N}^I_{bou} \cap \mathcal{N}^J_{bou}$ do
        $i = index^I(n)$
        $i_{bou} = index^{I,J}(n)$
        Construct the array $data\_send(i_{bou}) = data(i)$
        MPI_Isend to send the array $data\_send$ to $J$
        MPI_Irecv to receive the array $data\_receive^J$ from $J$
    end for
end for
```

4.4 matrix-vector and dot products

The two main parallel functions of an iterative solver, i.e., the matrix-vector and dot products, referred to as **MATVEC** and **DOTPRO** respectively, using the previously defined parallel operators **PAREXC** and **PARASM** are described here. The matrix-vector product in given by Algorithm 3. For the sake of clarity, the indices of the matrix are not given in the CSR format.

Algorithm 2 The parallel operator **PARASM** from Ω_I: data assembly

```
Input: a numeric array $data$ with length $\mathcal{N}^I$
Output: a modified array $data$
for each adjacent subdomain $J$ of $I$ do
    for each node $n \in \mathcal{N}^I_{bou} \cap \mathcal{N}^J_{bou}$ do
        $i = index^I(n)$
        $i_{bou} = index^{I,J}(n)$
        $data(i) = data(i) + data\_receive^J(i_{bou})$
    end for
end for
```

Algorithm 3 The parallel matrix-vector product **MATVEC**

```
for each subdomain $I$ do
    for each $n \in \mathcal{N}^I_{bou}$ do
        for each $a \in \mathcal{L}^I(n)$ do
            $i = index^I(a)$
            Construct $y^I(i) = 0$
            for each $b \in \mathcal{L}^I(n)$ do
                $j = index^I(b)$
                Construct $y^I(i) = y^I(i) + A^I(i,j) \times x^I(j)$
            end for
        end for
    end for
    Exchange: PAREXC($y^I$)
end for
for each subdomain $I$ do
    for each $n \in \mathcal{N}^I_{int}$ do
        for each $a \in \mathcal{L}^I(n)$ do
            $i = index^I(a)$
            Construct $y^I(i) = 0$
            for each $b \in \mathcal{L}^I(n)$ do
                $j = index^I(b)$
                Construct $y^I(i) = y^I(i) + A^I(i,j) \times x^I(j)$
            end for
        end for
    end for
end for
MPI_Waitall
for each subdoamin $I$ do
    Assemble: $y^I$ = PARASM($y^I$)
end for
```

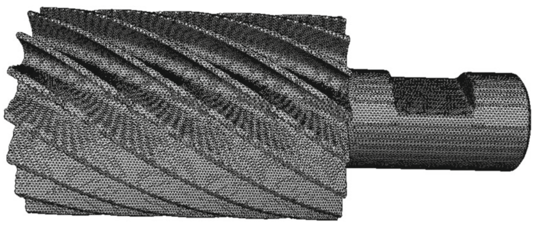

Fig. 8 Scalability test: computational mesh

Algorithm 4 describes the dot product. As mentioned previously, the loop over the nodes excludes those belonging to the set $\mathcal{N}^I_{bou,own}$ of each subdomain.

Algorithm 4 The parallel dot product **DOTPRO**

```
for each subdomain $I$ do
    $\alpha = 0$
    for each $n \in \mathcal{N}^I$ do
        $i = index^I(n)$
        if $n \in \mathcal{N}^I_{int}$ or $n \in \mathcal{N}^I_{bou,own}$ then
            $\alpha = \alpha + x^I(i) \times y^I(i)$
        end if
    end for
end for
MPI_AllReduce of $\alpha$
```

It must be emphasized that the functions **PAREXC** and **PARASM** are used to compute many other arrays during the execution the code, such as the construction of the mass matrix and the diagonal of the stiffness matrix to construct the inverse diagonal preconditioner.

4.5 Scalability test: shear stress of a milling cutter punch

This example, run in implicit, addresses the capability of Alya to deal with structures composed of millions of elements, while maintaining optimal scalability. The test consists in a linear elastic drill (with arbitrary mechanical properties) grasped in a shank. Gravity force and a punctual force along one of its lips are applied. The computational mesh is a regular mesh of 8.5 million tetrahedra, see Fig. 8. The tests have been run on MareNostrum cluster. The machine consists of 3,056 nodes, two 8-core processors (Inter Xeon E5-2670 cores at 2.6 GHz) per node and 32 GBytes of memory per node. Figure 9 shows the displacement (left figure) and the maximum and minimum shear stress after 20 time steps. Figure 10 shows the scalability obtained up to 1,024 CPUs. Note that the curve looses its optimallity at 1,024 CPUs, due to the small number of elements per CPU. In the next section the scalability with a finer mesh is studied.

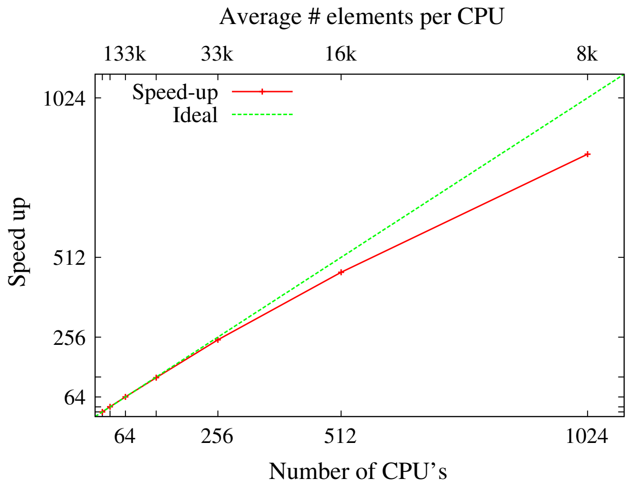

Fig. 10 Scalability test: scalability for a fixed number of iterations with a mesh of 8.5 million elements

5 Hybrid Based Strategy for Parallelization

The most important trends in contemporary computer architecture are currently leaning towards processors with multiple cores per socket with access to the same memory bank. This approach allows to run at lower frequencies with better overall performance than a single core processor. The parallelization strategy for this architecture is multithreading: a Master thread forks a number of Worker threads, and the computation is divided among them. The data communication and the synchronization between threads are done using the shared memory inside the multiprocessor. This strategy is known as parallelism at thread level and OpenMP is the standard interface for this model [11].

As detailed in the previous section, parallelization in Alya is mainly done via MPI. The domain decomposition strategy implemented only uses parallelism at task level, which

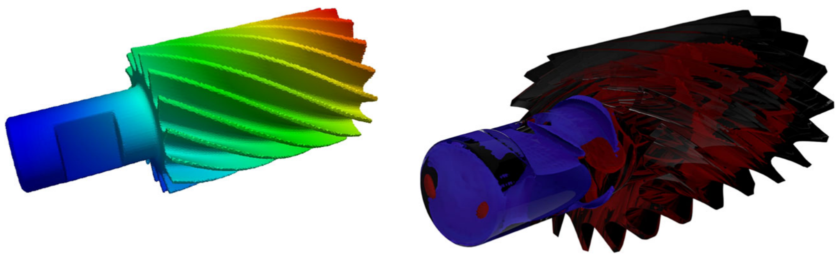

Fig. 9 Scalability test: displacement field (*left*) and maximum and minimum shear stress field (*right*). The minimum value is shown in *blue* and the maximum in *red*. (Color figure online)

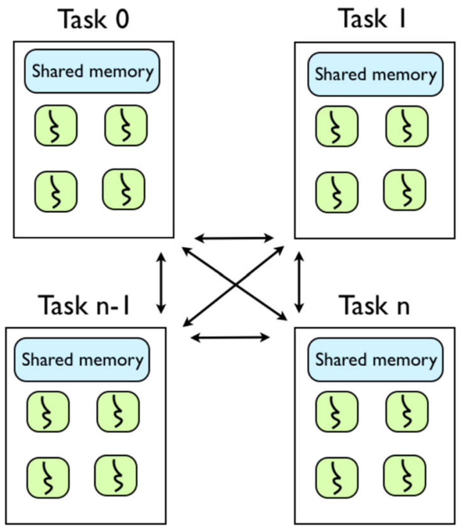

Fig. 11 Multicore architecture for an hybrid openMP/MPI framework

Table 1 Routines costs in terms of the sum of time of the function, including the time of the called subfunctions with respect to the total execution time in the explicit case

Cost (%)	Routine	Description
…	…	
93,6	Iterative scheme	Controlling the internal loop of the equations
93,6	Explicit time integration	Explicit time stepping scheme (Newmark)
93,5	Iterative solution	Solving an iteration of the equations
93,5	Matrix	Computing the elemental matrix and RHS
93,5	Assembly	Assembling the matrix and RHS
36,4	Finite element computations	Computing derivatives at gauss points
…	…	

is provided by MPI. For that reason, a hybrid code with OpenMP/MPI is developed in order to take advantage of all levels of parallelism that a multicore architecture offers and also to enable one MPI task to access all the memory of a node. The main feature of this multicore-architecture relies on the fact that both the shared memory inside the processor and the message passing between nodes are leveraged. The structure consists of two steps: first, it assigns one task to each node and secondly, the node creates a thread per core, see Fig. 11. An important reduction of the communication cost between cores is then expected.

5.1 Introducing OpenMP into the MPI Parallel Code

To exploit the thread-level parallelism of a multicore architecture, OpenMP directives are added to the MPI parallel code. The first step consists in selecting the most-time consuming routines of the code in total execution time. Among these routines, not all of them are susceptible to be parallelized at thread-level: they must contain a loop structure with independent iterations. The targeted loops must contain a large body of code, since a considerable amount of computational work hides behind the overhead of thread-management.

Table 1 depicts the most time-consuming routines of Alya in terms of the total execution time in the explicit case. Note that the cost refers to the percentage of the total execution time spent, also including the call to other subroutines. In order to select the proper subroutine to be optimized by the parallelization with OpenMP, it is important to identify the calls between them. For instance, the routine that computes the matrix costs the same as the assembly one, which means that the time spent by the matrix construction routine itself is minimum, since it has a call to the assembly subroutine. Hence, the assembly routine is the one that has to be optimized. In the FE matrix and RHS assembly routine, the discrete system to be solved is formed by looping over all the elements of the mesh, adding the contributions from that element to the global matrix and RHS. This routine is expensive for a number of reasons: significant loop nesting, many matrices assembling and several calls to other subroutines (for instance, to constitutive models).

In order to introduce thread parallelism in the assembly loop, two main OpenMP directives are used to indicate the compiler how to parallelize appropriately:

– *Guided scheduling*: since the workload within each iteration is not the same, iterations are assigned to threads in blocks. In a guided scheduling the block size decreases within each iteration (in contrast with the dynamic scheduling), thus obtaining a better relation between the thread management time and the balanced workload.
– *Data scope*: the variables that are shared among the iterations are visible to all threads, while the ones that have an independent value among iterations have a different copy per thread.

A critical point on the thread parallelism relies on the so called *race conditions*, which might lead to non-deterministic results. Race conditions arise when different threads try to update the same state or array at the same time. Code paths accessing and manipulating shared data simultaneously are known as *critical regions*. It is clear that both the operation to assemble an element into a local matrix and the addition of that local matrix into the global matrix must be thread safe. Even though these regions cannot be fully avoided it is

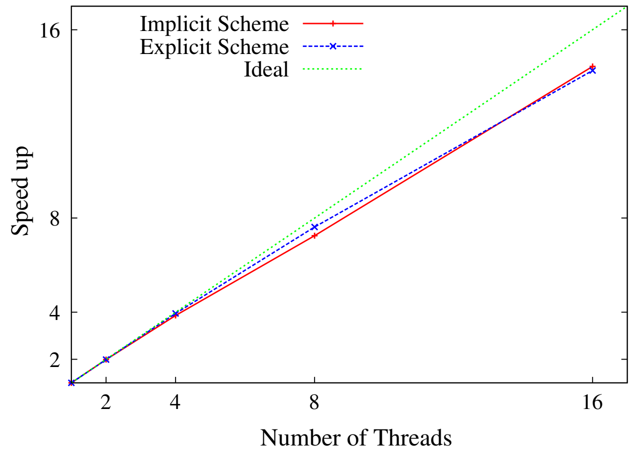

Fig. 12 Scalability of the assembly routine parallelized with OpenMP in Alya-*solidz*

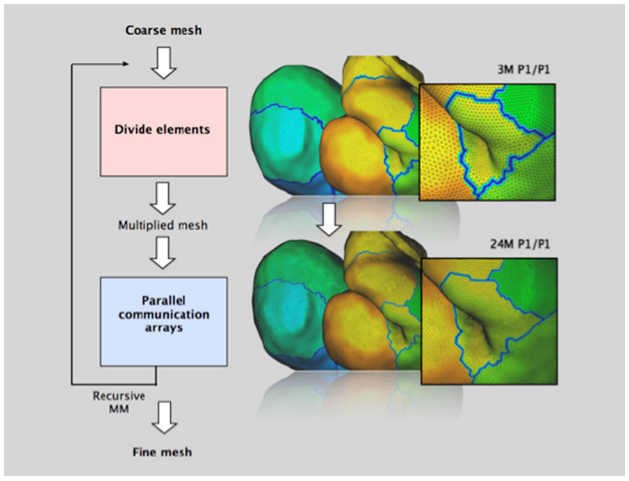

Fig. 13 Example 1: Mesh multiplication algorithm

important to minimize their occurrence, because their abuse can serialize the execution, hence losing the optimal parallel architecture of the code. One possible solution is to specify different region names, combined with the use of *atomic* clauses (uninterruptible instructions) for a single memory location.

Such scheme was implemented in Alya, paying particular attention to the specificities of solid mechanics codes. In order to evaluate the gain in terms of time execution that OpenMP architectures provides, a reduced case was computed. As an illustration of the strategy used here for all identified subroutines, Fig. 12 depicts the speedup analysis of the assembly routine for both explicit and implicit schemes. The performance with one thread is the same as the one for the sequential version and, the thread version for both schemes is speeding up quasi-optimally up to at least 16 cores.

6 Numerical Examples

This section presents several examples of different nature with the purpose of showing the applicability, scalability, performance and flexibility of Alya when solving a solid mechanics problems.

6.1 Example 1: Mesh Multiplication

In petascale applications, the pre- and post-process tasks are becoming a bottleneck in the complete simulation cycle. Techniques such as parallel I/O have been introduced to mitigate these effects in post-processing, but these are only effective within a limited range. Mesh Multiplication (MM) was introduced as an alternative [47]. This technique consists in refining the mesh uniformly, recursively, on-the-fly and in parallel. For tetrahedra, hexahedra and prisms, each level multiplies the number of elements by eight, while a pyramid is divided into ten new elements. Apart from generating a fine mesh in parallel, the MM strategy enables to carry out the simulation on this fine mesh without having to permanently store it at anytime during the simulation process.This technique is also very useful for studying mesh convergence as well as weak or strong scalability [47]. Figure 13 shows the recursive MM algorithm.

The example consists of a linear elastic large structure with arbitrary mechanical properties under its own weight and magnetic load. The initial mesh is composed of a mesh of 491,415 tetrahedral elements. As an example of the efficiency of the algorithm, a mesh of 250 million elements was also obtained in just 1 second on 8,000 CPUs. Figure 14 shows the original mesh and the obtained after MM. Figure 15 depicts the displacement field after 200 iterations using an implicit scheme and the hybrid version of Alya.

Figure 16 shows the speedup obtained with a mesh of 32 million elements (2-level mesh) from 64 to 2,048 CPUs. The code shows optimal scalability when solving mechanical problems. Note that in contrast with the drill problem, the amount of elements per core is significatively larger.

6.2 Example 2: The Chimera Method—Two-Material Cube

As an illustration of the flexibility of the Alya-*solidz* module, the Chimera method is applied to the solid mechanics equations described in Sect. 2.1. This family of methods allows to handle non-conforming and overlapping meshes, thus simplifying the construction of computational meshes of complex geometries.

The origin of the Chimera Method is found in the context of Computational Fluid Dynamics. It was originally developed by Steger and coworkers [22,75,76]. It consists in superimposing an independent mesh referred to as the

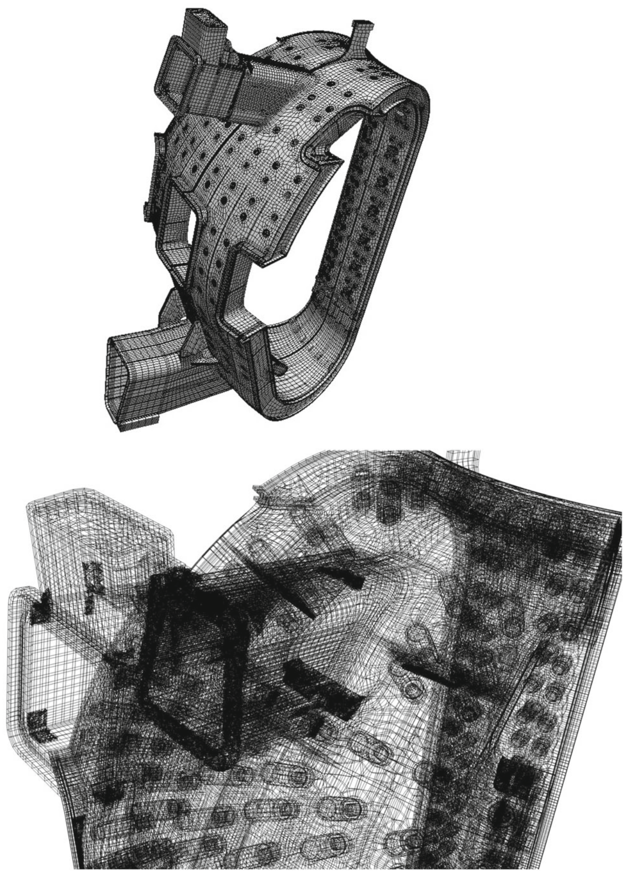

Fig. 14 Example 1: Initial mesh of the structure (*left*) and detail of the mesh after mesh multiplication (*right*). Fusion reactor vacuum vessel. Geometry and mesh property of F4E

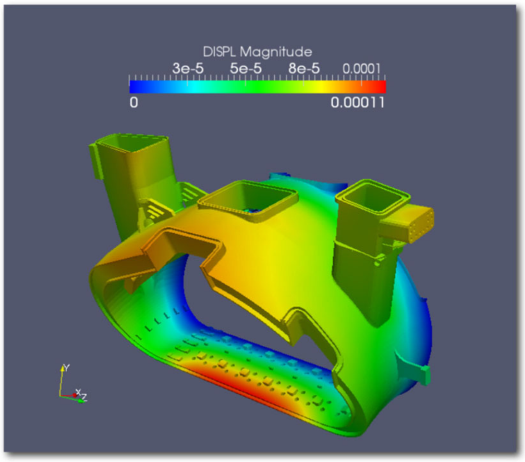

Fig. 15 Example 1: Displacement field after 200 iterations using and implicit scheme

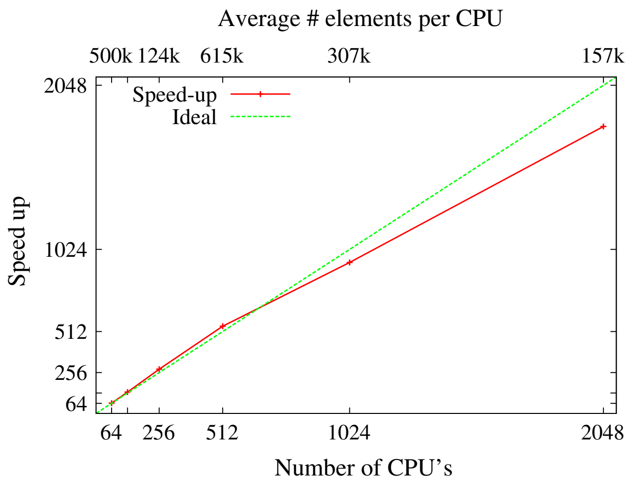

Fig. 16 Example 1: Scalability of the fusion reactor problem with a mesh of 32 million elements

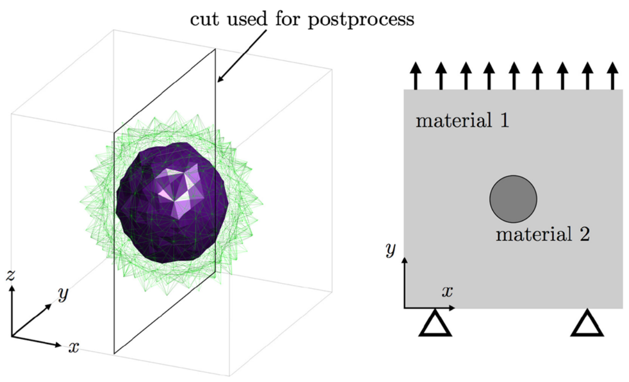

Fig. 17 Example 2: Extension elements and hole (*left*) and boundary conditions (*right*)

patch mesh on a larger mesh covering the overall computational domain and called the *background mesh*. In Alya, the patch mesh and its corresponding constitutive properties are responsible for the mechanical deformation of the body it covers, while the mechanics of the rest of the body is defined by the part of the background mesh not covered by the patch mesh. The coupling between the boundary of the patch mesh and the background mesh is then enforced by additional *transmission conditions*. In the framework of solid mechanics, this non-local approach allows for the independent meshing of complex intertwined geometries without being constrained by mesh conformity conditions at their boundaries. See Ref. [29] for more details.

The Chimera method is applied here to the solution of a 3D example shown in Fig. 17. The geometry is composed of two different Neo-Hookean materials, one of which corresponds to the spherical patch mesh enclosed in a second material. In this example arbitrary constitutive parameters have been considered for both material, but material 2 (within the sphere) is stiffer than material 1. The cube is submitted to an increasing y-displacement on the top face, the bottom face being constrained in the y-direction. Additional lateral boundary conditions are applied to avoid rigid body motion (not shown for clarity).

Table 2 Example 2: Number of elements and mesh size

	Elements total	Elements extension	Elements hole	Elements size
Mesh 1	374,848	14,848	17,120	0.0029
Mesh 2	2,939,664	59,664	151,448	0.00036
Reference mesh	10,790,400			0.000093

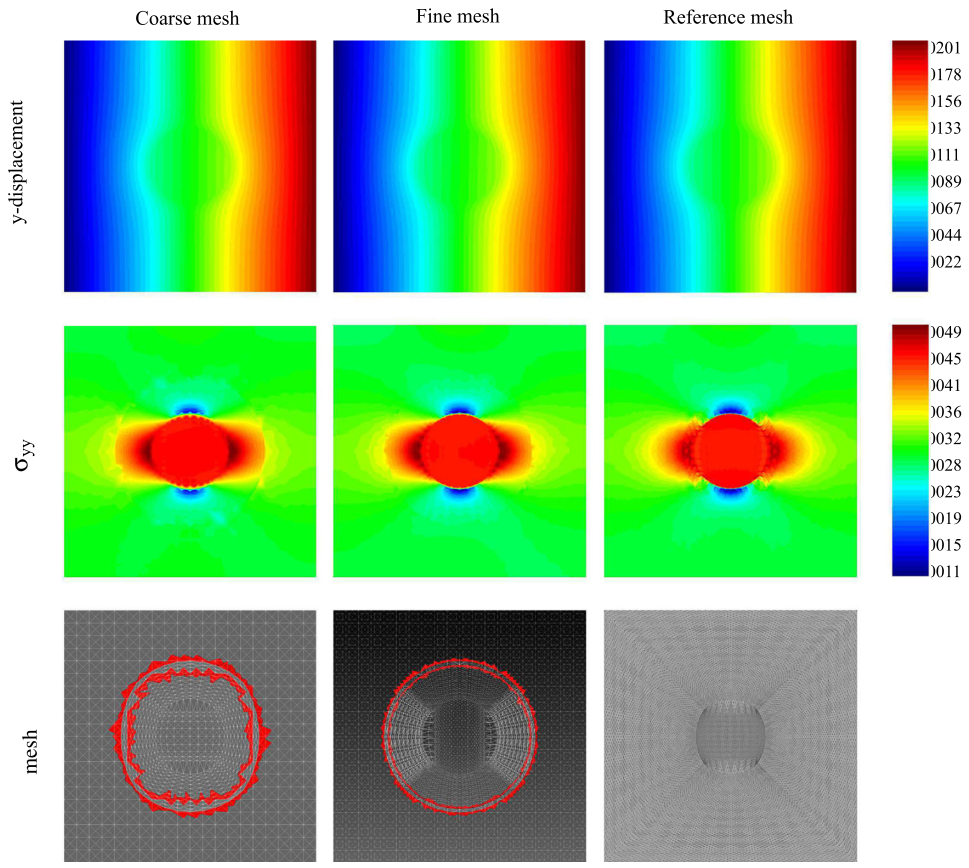

Fig. 18 Example 2: Solution on mid yz-plane for the three meshes: y-displacement (*top*) and stress along the y-direction (*middle*) and computational mesh (*bottom*)

The problem was solved on two different meshes with the Chimera method: a coarse one of 400,000 elements and a fine one of 3 million elements. A reference solution without the Chimera method was also computed in a refined mesh of 10 million elements. Table 2 shows the geometrical details of the three different meshes. Figure 18 shows the displacement and stress in the y direction as well as the corresponding mesh obtained on the mid yz-plane for the coarse and fine meshes of the Chimera method. For the Chimera meshes, the extension elements used to couple the patch and background mesh are delimited in red. Figure 19 shows the solution for an additional cut along the vertical displacement and stress. The solution on both meshes with the Chimera method is compared with the reference one. A good qualitative agreement is observed for the displacement even for the larger mesh. The stress exhibits a lower convergence, but this should be tempered by the fact that the meshes are only first order for the solution derivatives. In the overlapping zone, covered by the extension elements, two solutions coexist: the one obtained on the patch and the one obtained on the background. The discontinuity in the stress inside the overlapping zone is appreciable for the coarsest mesh, but this jump decreases when refining the mesh. Nevertheless, the maximum error between the solution with Chimera computed with the fine mesh and the reference solution is of order less than 10^{-4}.

6.3 Example 3: Coupled Electro-mechanical Model of the Heart

In this case, Alya is used to simulate the cardiac ventricular contraction, showing the potential for simulating coupled problems. See Ref. [56,80] for a complete discussion of the methodology. The heart is made of elongated cells called myocites arranged as a compact fibered helicoidal structure. As an electrical impulse propagates, the fibers contract longitudinally making the heart pump the blood out of its cavities thanks to the fiber arrangement. At organ level, tissue is con-

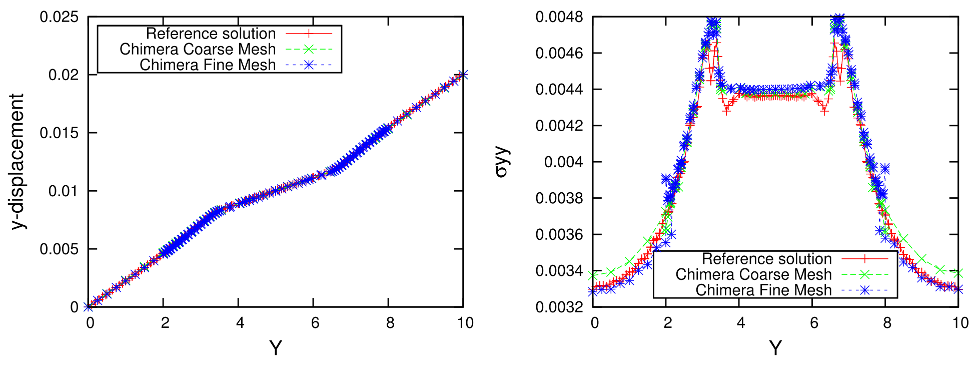

Fig. 19 Example 2: Solution along mid line of mid yz-plane for the three meshes: y-displacement (*left*) and stress along the y-direction (*right*)

sidered as a continuum composite material with anisotropic behaviour.

The cardiac computational model can be decomposed into three parts:

– *Electrophysiology:* Action potential propagation $\phi(x_i, t)$ is modelled through a transient anisotropic diffusion equation with a non-linear reaction term:

$$\frac{\partial \phi}{\partial t} = \frac{\partial}{\partial x_i}\left(D_{ij}\frac{\partial \phi}{\partial x_j}\right) + L(\phi) \tag{19}$$

Diffusion is governed by the tensor D_{ij}, which represents the fiber orientation at each physical point. Its diagonal components are the axial and crosswise fibre diffusion. The crosswise diffusion is around one third of the axial diffusion. Finally, $L(\phi_\alpha)$ is the non-linear reaction term corresponding to the cell model. Depending on the model used, this term ranges from a simple cubic equation to an ordinary differential equation system of one hundred equations coupled and solved simultaneously. This term models the ionic currents interaction behind the muscular cellular mechanisms.

– *Mechanical deformation:* From a mechanical point of view, cardiac tissue is a thick layered structure: endo-, epi- and myocardium. The material is compressible hyperelastic, with anisotropic behaviour ruled by the fibre structure. The composite character of the material is determined by the internal stress, which is developed in two parts, active and passive:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}_{pas} + \sigma_{act}(\lambda, [Ca^{2+}])\boldsymbol{f} \otimes \boldsymbol{f} \tag{20}$$

The passive part is governed by a transverse isotropic exponential strain energy function $W(b)$ that relates the Cauchy stress σ to the right Cauchy-Green deformation $\boldsymbol{b}$ [56]:

$$J\boldsymbol{\sigma}_{pas} = (a\ e^{b(I_1-3)} - a)\boldsymbol{b} + 2a_f(I_4-1)e^{b_f(I_4-1)^2} \\ \boldsymbol{f} \otimes \boldsymbol{f} + K(J-1)\boldsymbol{I} \tag{21}$$

The strain invariant I_1 represents the non-collagenous material while strain invariant I_4 represents the stiffness of the muscle fibers, and a, b, a_f, b_f are parameters to be determined experimentally. K is the bulk modulus and $\boldsymbol{f}$ defines the fibre direction. The active part comes through the electro-mechanical coupling and is described as follows.

– *Electro-mechanical Coupling:* The contracting electrical component of the electro-mechanical coupling is initiated almost simultaneously in several zones of the ventricular epicardium. Cardiac mechanical deformation is the result of the active tension generated by the myocytes. The model assumes that the active stress is produced only in the direction of the fibre and depends on the calcium concentration of the cardiac cell:

$$\sigma_{act} = \gamma\ \frac{[Ca^{2+}]^n}{[Ca^{2+}]^n + C_{50}^n}\sigma_{max}(1 + \beta(\lambda_f - 1)). \tag{22}$$

In this equation, C_{50}^n, σ_{max}, λ_f and $0 < \gamma < 1$ are model parameters.

Both electrophysiology and mechanical action is simulated in Alya on the same mesh, as fine as the case demands. The time integration scheme is a staggered, with both problems solved explicitly. The fibre fields can come either from measurements (using a so-called Diffusion Tensor MRI technique) or from semi-empirical rule-based models [56].

Figure 20 shows several snapshots of a bi-ventricular geometry during systole. Figure 21 shows the evolution of the ejection rate, which represents the heart pumping action, measuring the change in the ventricular cavities volume.

6.4 Example 4: Fluid–Structure Interactions—Wall's Problem

This problem [84] is proposed to demonstrate the ability to solve FSI problems with complex flow-flexible structure interactions exhibiting large deformations.

The problem consists of a thin elastic nonlinear shell attached to a fixed square rigid body, which are submerged

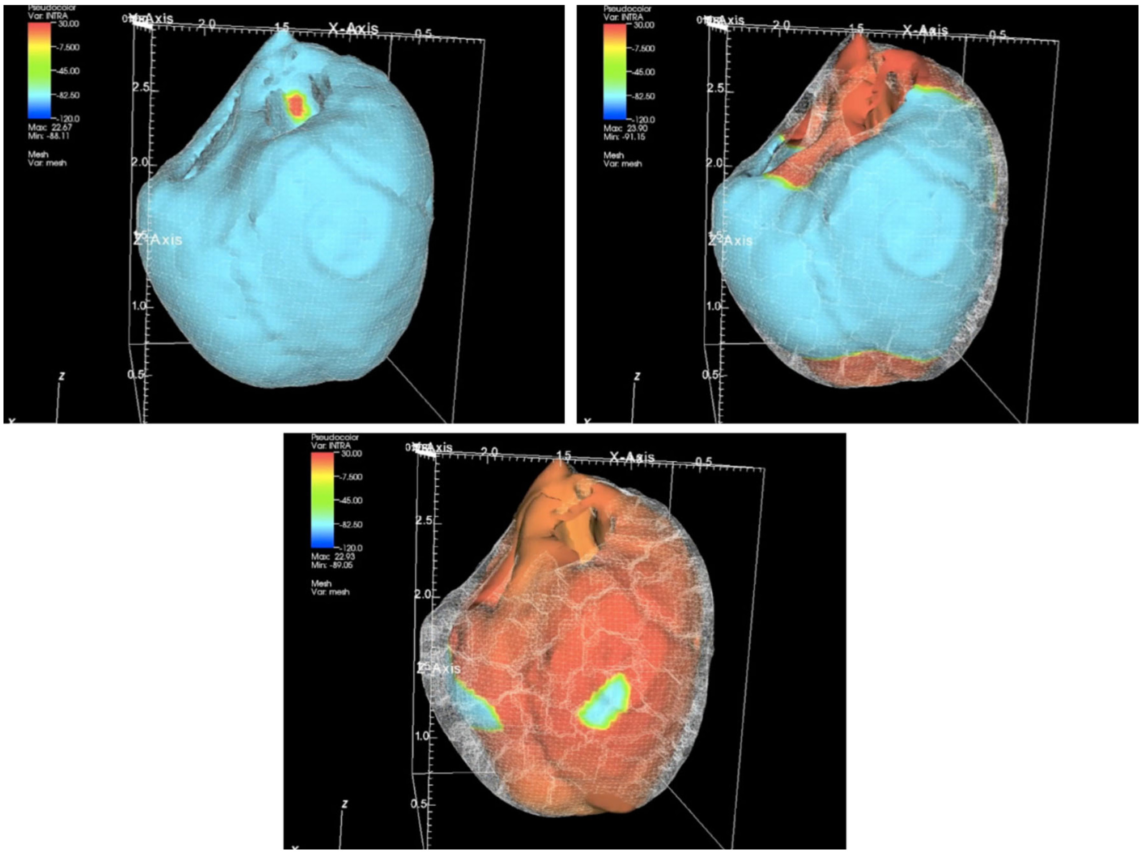

Fig. 20 Example 3: Bi-ventricular electrical activation propagation of a dog heart during systole

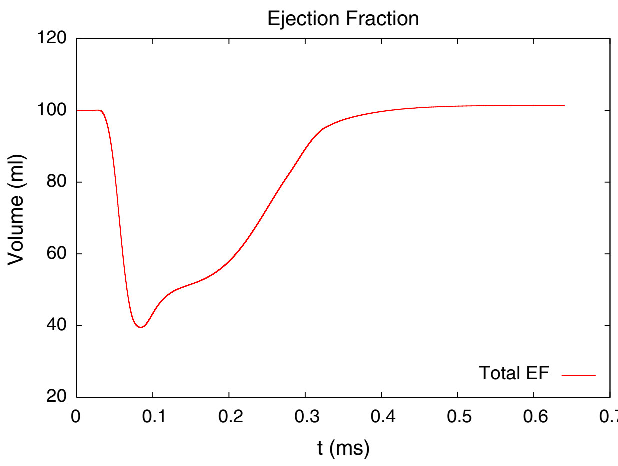

Fig. 21 Example 3: Heart ejection rate

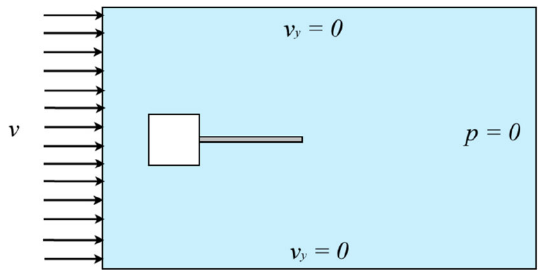

Fig. 22 Example 4: Boundary conditions

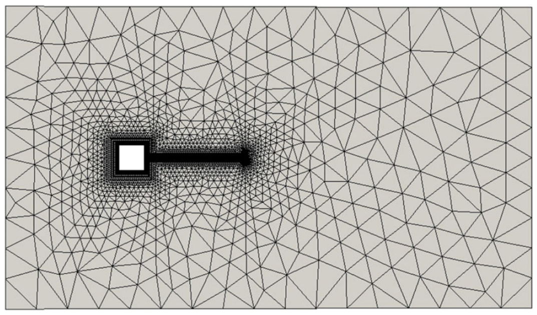

Fig. 23 Example 4: reference finite element mesh

in an incompressible fluid flow. Both media have arbitrary mechanical properties but such that the Reynolds number is $Re = 204$ if the length of the square rigid body is taken as the characteristic length. Inflow boundary conditions are imposed on the left wall of the fluid domain, outflow on the right wall and slip boundary conditions at the top and bottom

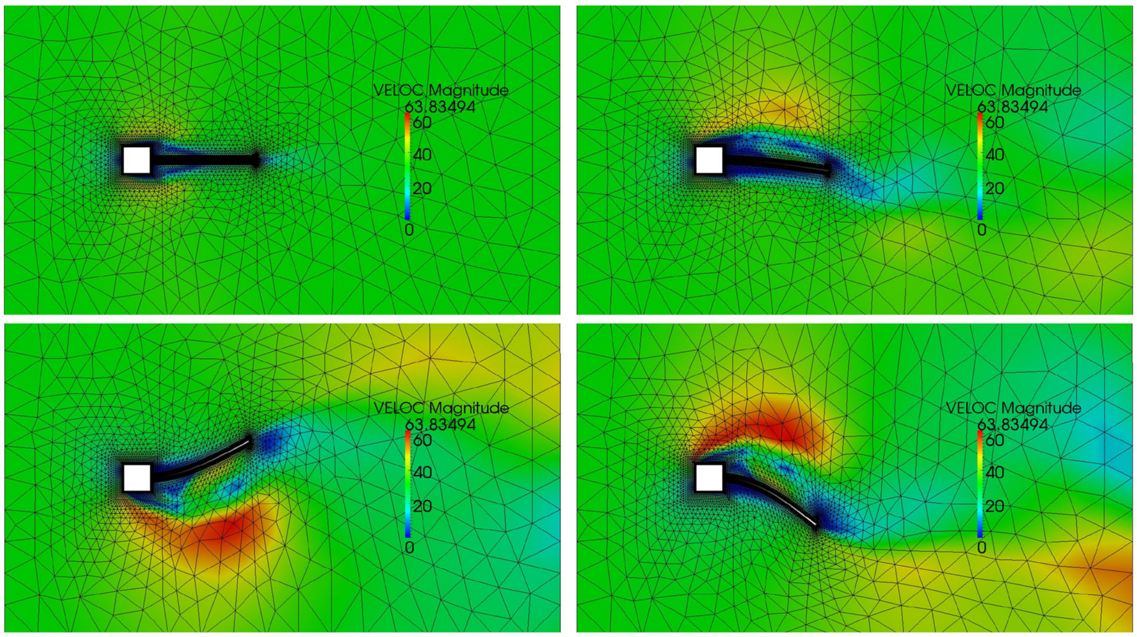

Fig. 24 Example 4: fluid velocity field for different times

wall, see Fig. 22. Non-slip boundary conditions are imposed along the body and the structure. Again, both problems are solved using Alya.

The finite element mesh is shown in Fig. 23. It consists of a hybrid mesh of 6,919 elements, composed of tetrahedra and quadrilateral elements. The fluid domain contains 6,695 elements, while the solid one has 224 elements.

A weak coupling scheme [15,78] was used to couple the fluid with the structure. The key point of the strategy implemented in Alya is to divide geometrically the problem in non-overlapping fluid and solid zones, such that by properly distributing the amount of processors working within each zone, the amount of computations performed by the CPUs is optimized.

Figure 24 shows the structural displacement at different times $t = 0, 3.58, 7.01$ and $15.58s$ along with the fluid velocity. Note that the mesh is adapted to the solid structure displacement, since the ALE formulation used here computes the convective velocity of the fluid according to the difference between the material and mesh velocities. The results can be compared to the ones obtained in Ref. [84].

7 Conclusion

This paper presents the solid mechanics module of Alya code for solving linear and non-linear continua problems with large deformations addressing also the potential for solving complex coupled problems in parallel. The parallelization strategy used in Alya is formally described in the context of MPI communications. To enhance the massively parallel performance of the code, OpenMP directives were added to the current MPI Alya code. The code was tested successfully using thousands of CPUs to solve problems with up to millions of elements. The flexibility of solid module in Alya was demonstrated by using more complex preprocessing techniques, such as a mesh multiplication algorithm and the Chimera method. Those were successfully tested with three-dimensional structural tests showing optimal results. Examples of FSI and electro-mechanical coupling were also shown, demonstrating the capability of the code for solving multi-physic problems with high-performance computing. Future work is aiming at implementing more complex FE techniques and overcoming the barrier of more than 100,000 CPUs.

Acknowledgments D.D.T and A.J acknowledge funding through SIMUCOMP and ERA-NET MATERA+ project financed by the Consejería de Educación y Empleo of the Comunidad de Madrid and by the European Union's Seventh Framework Programme (FP7/2007-2013). This work was partially supported by the grant SEV-2011-00067, Severo Ochoa Program, awarded by the Spanish Government. The authors' would like to acknowledge PRACE infrastructure support.

References

1. Alya system. http://www.bsc.es/computer-applications/alya-system
2. Bigdft. http://bigdft.org/Wiki/index.php?title=Presenting_BigDFT
3. Codeaster. http://www.code-aster.org/
4. CODE_SATURN. http://code-saturne.org

5. Febio. http://febio.org/
6. Metis, family of multilevel partitioning algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis
7. Openacc. https://developer.nvidia.com/openacc
8. Opencl. https://developer.nvidia.com/opencl
9. Openfoam. http://www.openfoam.com/
10. Openmp. http://openmp.org/
11. Openmp. http://openmp.org/wp/
12. Summary of available software for sparse direct methods. http://www.cise.ufl.edu/research/sparse/codes/
13. Adams M (1999) Parallel multigrid algorithms for unstructured 3d large deformation elasticity and plasticity finite element problems. Technical report UCB/CSD-99-1036, EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/5398.html
14. Adams MF (2007) Algebraic multigrid methods for direct frequency response analyses in solid mechanics. Comput Mech 39:497–507
15. Malan AG, Oxtoby O (2013) An accelerated, fully-coupled, parallel 3d hybrid finite-volume fluid–structure interaction scheme. Comput Methods Appl Mech Eng 253:426–438
16. Amestoy P, Duff I, L'Excellent JY (2000) Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput Methods Appl Mech Eng 184(2–4):501–520
17. Arbenz P, van Lenthe G, Mennel U, Müller R, Sala M (2008) A scalable multi-level preconditioner for matrix-free μ-finite element analysis of human bone structures. Int J Numer Methods Eng 73:937–947. doi:10.1002/nme.2101
18. Badia S, Martin A, Principe J (2014) A highly scalable parallel implementation of balancing domain decomposition by constraints. SIAM J Sci Comput 36(2):C190–C218
19. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zhang H (2014) PETSc Web page. http://www.mcs.anl.gov/petsc
20. Balay S, Adams MF, Brown J, Brune P, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zhang H (2013) PETSc users manual. Technical report ANL-95/11—revision 3.4, Argonne National Laboratory. http://www.mcs.anl.gov/petsc
21. Becker G, Noels L (2013) A full-discontinuous galerkin formulation of nonlinear kirchhoff-love shells: elasto–plastic finite deformations, parallel computation, and fracture applications. Int J Numer Methods Eng 93:80–117. doi:10.1002/nme.4381
22. Benek J (1986) Chimera. A grid-embedding technique. Technical report, DTIC Document
23. Bhardwaj M, Pierson K, Reese G, Walsh T, Day D, Alvin K, Peery J, Farhat C, Lesoinne M (2002) Salinas: a scalable software for high-performance structural and solid mechanics simulations. In: Proceedings of the 2002 ACM/IEEE conference on supercomputing., SC'02IEEE Computer Society Press, Los Alamitos, pp 1–19
24. Blatt M (2009) Dune on bluegeen/p. In: Proceedings of 15th SciComp. University Press
25. Casasdei F, Avotins J (1997) A language for implementing computational mechanics applications. In: Technology of object-oriented languages and systems, 1997. TOOLS 25, Proceedings, pp 52–67
26. Ciccozzi F (2013) Towards code generation from design models for embedded systems on heterogeneous cpu-gpu platforms. In: 2013 IEEE 18th conference on emerging technologies factory automation (ETFA), pp 1–4
27. Cirak F, Deiterding R, Mauch S (2007) Large-scale fluidstructure interaction simulation of viscoplastic and fracturing thin-shells subjected to shocks and detonations. Comput Struct 85:1049–1065. doi:10.1016/j.compstruc.2006.11.014
28. Duff I, Reid J (1983) The multifrontal solution of indefinite sparse symmetric linear-equations. ACM Trans Math Softw 9(3):302–325
29. Eguzkitza B, Houzeaux G, Aubry R, Owen H, Vázquez M (2013) A parallel coupling strategy for the Chimera and domain decomposition methods in computational mechanics. Comput Fluids 80:128–141
30. El maliki A, Fortin M, Tardieu N, Fortin A (2010) Iterative solvers for 3d linear and nonlinear elasticity problems: Displacement and mixed formulations. Int J Numer Methods Eng 83(13):1780–1802
31. Falgout RD, Yang UM (2002) hypre: a library of high performance preconditioners. In: Preconditioners, lecture notes in computer science, pp 632–641
32. Farhat C, Roux FX, Oden JT (1994) Implicit parallel processing in structural mechanics. Elsevier Science SA, Amsterdam
33. Flaig C, Arbenz P (2011) A scalable memory efficient multigrid solver for micro-finite element analyses based on CT images. Parallel Comput 37:846–854. doi:10.1016/j.parco.2011.08.001
34. George A, Liu JW (1981) Computer solution of large sparse positive definite. Prentice Hall, Englewood cliffs (Professional technical reference)
35. Gerstenberger A, Tuminaro R (2013) An algebraic multigrid approach to solve extended finite element method based fracture problems. Int J Numer Methods Eng 94:248–272. doi:10.1002/nme.4442
36. Geuzaine C, Remacle JF (2009) Gmsh: a 3-d finite element mesh generator with built-in pre-and post-processing facilities. Int J Numer Methods Eng 79(11):1309–1331
37. Goudreau G, Hallquist J (1982) Recent developments in large-scale finite element Lagrangian hydrocode technology. Comput Methods Appl Mech Eng 33:725–757. doi:10.1016/0045-7825(82)90129-3
38. Gould NIM, Scott JA, Hu Y (2007) A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. ACM Trans Math Softw 33(2):300–331
39. Grinberg L, Pekurovsky D, Sherwin SJ, Karniadakis GE (2009) Parallel performance of the coarse space linear vertex solver and low energy basis preconditioner for spectral/hp elements. Parallel Comput 35(5):284–304
40. Gupta A, Koric S, George T (2009) Sparse matrix factorization on massively parallel computers. In: Proceedings of the conference on high performance computing networking, storage and analysis, SC'09, ACM, New York, pp 1:1–1:12
41. Goddeke D, Wobker H, Strzodka R, Mohd-Yusof J, McCormick P, Turek S (2009) Co-processor acceleration of an unmodified parallel solid mechanics code with feastgpu. Int J Comput Sci Eng 4(4):254–269
42. Hales JD, Novascone SR, Williamson RL, Gaston DR, Tonks MR (2012) Solving nonlinear solid mechanics problems with the Jacobian-free Newton Krylov method. Comput Model Eng Sci 84:84–123
43. Heath M, Ng E, Peyton B (1991) Parallel algorithms for sparse linear systems. SIAM Rev 33(3):420–460
44. Heil M, Hazel A, Boyle J (2008) Solvers for large-displacement fluid–structure interaction problems: segregated versus monolithic approaches. Comput Mech 43(1):91–101
45. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, Williams A, Stanley KS (2005) An overview of the Trilinos project. ACM Trans Math Softw 31(3):397–423
46. Hess B, Kutzner C, van der Spoel D, Lindahl E (2008) GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. J Chem Theory Comput 4(3), 435–447 (2008) doi:10.1021/ct700301q
47. Houzeaux G, de la Cruz R, Owen H, Vázquez M (2013) Parallel uniform mesh multiplication applied to a Navier–Stokes solver. Comput Fluids 80:142–151

48. Houzeaux G, Vázquez M, Aubry R, Cela J (2009) A massively parallel fractional step solver for incompressible flows. JCP 228(17):6316–6332
49. Hughes T, Ferencz R (1987) Large-scale vectorized implicit calculations in solid mechanics on a CrayX-MP/48 utilizing EBE preconditioned conjugate gradients. Comput Methods Appl Mech Eng 61:215–248. doi:10.1016/0045-7825(87)90005-3
50. Hughes TJ (2012) The finite element method: linear static and dynamic finite element analysis. DoverPublications.com
51. Hussain M, Abid M, Ahmad M, Khokhar A, Masud A (2011) A parallel implementation of ALE moving mesh technique for FSI problems using OpenMP. Int J Parallel Program 39:717–745. doi:10.1007/s10766-011-0168-3
52. Jouglard C, Coutinho A (1998) A comparison of iterative multilevel finite element solvers. Comput Struct 69(5):655–670
53. Kilic SA, Saied F, Sameh A (2004) Efficient iterative solvers for structural dynamics problems. Comput Struct 82(28):2363–2375
54. Knoll D, Keyes D (2004) Jacobian-free Newton Krylov methods: a survey of approaches and applications. J Comput Phys 193(2):357–397
55. Komatitsch D, Erlebacher G, Göddeke D, Michéa D (2010) High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. J Comput Phys 229:7692–7714. doi:10.1016/j.jcp.2010.06.024
56. Lafortune P, Arís R, Vázquez M, Houzeaux G (2012) Coupled electromechanical model of the heart: parallel finite element formulation. Int J Numer Methods Biomed Eng 28:72–86. doi:10.1002/cnm.1494
57. Lang S, Wieners C, Wittum G (2002) The application of adaptive parallel multigrid methods to problems in nonlinear solid mechanics. In: Ramm E, Rank E, Rannacher R, Schweizerhof K, Stein E, Wendland W, Wittum G, Wriggers P, Wunderlich W (eds) Error-controlled adaptive finite elements in solid mechanics, 422 pp, ISBN: 978-0-471-49650-2
58. Li X, Demmel JW (2003) Superlu dist: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans Math Softw 29:110–140
59. Liu WK, Belytschko T, Moran B (2000) Nonlinear finite elements for continua and structures. Wiley, New York
60. Liu Y, Zhou W, Yang Q (2007) A distributed memory parallel element-by-element scheme based on Jacobi-conditioned conjugate gradient for 3D finite element analysis. Finite Elem Anal Design 43:494–503. doi:10.1016/j.finel.2006.12.007
61. Logg A, Mardal KA, Wells GN (eds) (2012) Automated solution of differential equations by the finite element method. Lecture notes in computational science and engineering, vol 84. Springer, Berlin. doi:10.1007/978-3-642-23099-8
62. Lohner R, Mut F, Cebral J, Aubry R, Houzeaux G (2010) Deflated preconditioned conjugate gradient solvers for the pressure-poisson equation: Extensions and improvements. Int J Numer Meth Eng 10196–10208
63. Luebke D (2008) Cuda: scalable parallel programming for high-performance scientific computing. In: 5th IEEE international symposium on biomedical imaging: from nano to macro, 2008. ISBI 2008, pp 836–838
64. Maday Y, Magoulès F (2006) Absorbing interface conditions for domain decomposition methods: a general presentation. Comput Methods Appl Mech Eng 195(29–32):3880–3900
65. Maurer D, Wieners C (2011) A parallel block LU decomposition method for distributed finite element matrices. Parallel Comput 37(12):742–758
66. McCormick SF (1987) Multigrid methods. Frontiers in applied mathematics. Philadelphia, Pa. Society for Industrial and Applied Mathematics
67. Moore D, Jérusalem A, Nyein M, Noels L, Jaffee M, Radovitzky R (2009) Computational biology—modeling of primary blast effects on the central nervous system. NeuroImage 47(2):T10–T20. doi:10.1016/j.neuroimage.2009.02.019
68. Owens J, Houston M, Luebke D, Green S, Stone J, Phillips J (2008) Gpu computing. Proc IEEE 96(5):879–899
69. Paszyski M, Jurczyk T, Pardo D (2013) Multi-frontal solver for simulations of linear elasticity coupled with acoustics. Comput Sci 12(0). http://journals.agh.edu.pl/csci/article/view/102
70. Quey R, Dawson PR, Barbe F (2011) Large-scale 3D random polycrystals for the finite element method: Generation, meshing and remeshing. Comput Methods Appl Mech Eng 200:1729–1745. doi:10.1016/j.cma.2011.01.002
71. Radovitzky R, Seagraves A, Tupek M, Noels L (2011) A scalable 3D fracture and fragmentation algorithm based on a hybrid, discontinuous galerkin, cohesive element method. Comput Methods Appl Mech Eng 200:326–344. doi:10.1016/j.cma.2010.08.014
72. Saad Y (2003) Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics
73. Smith BF (1995) Domain decomposition methods for partial differential equations. In: Proceedings of ICASE/LaRC Workshop on Parallel Numerical Algorithms. University Press
74. Soto O, Löhner R, Camelli F (2003) A linelet preconditioner for incompressible flow solvers. Int J Numer Meth Heat Fluid Flow 13(1):133–147
75. Steger J, Benek FDJ (1983) A chimera grid scheme. Adv Grid Gener 5:59–69
76. Steger J, Benek J (1987) On the use of composite grid schemes in computational aerodynamics. Comput Meth Appl Mech Eng 64:301–320
77. Stewart JR, Edwards H (2004) A framework approach for developing parallel adaptive multiphysics applications. Finite Elem Anal Design 40(12):1599–1617 (The Fifteenth Annual Robert J. Melosh Competition)
78. Kalro V, Tezduyar TE (2000) A parallel 3d computational method for fluid–structure interactions in parachute systems. Comput Methods Appl Mech Eng 190:1467–1482
79. van Rietbergen B, Weinans H, Huiskes R, Polman B (1996) Computational strategies for iterative solutions of large FEM applications employing voxel data. Int J Numer Methods Eng 39:2743–2767. doi:10.1002/(SICI)1097-0207(19960830)39:162743:AID-NME9743.3.CO;2-1
80. Vázquez M, Arís R, Houzeaux G, Aubry R, Villar P, Garcia-Barnós J, Gil D, Carreras F (2011) A massively parallel computational electrophysiology model of the heart. Int J Numer Methods Biomed Eng 27(12):1911–1929. doi:10.1002/cnm.1443
81. Vázquez M, Houzeaux G, Grima R, Cela J (2007) Applications of parallel computational fluid mechanics in MareNostrum supercomputer: low-mach compressible flows. In: PARCFD2007. Antalya (Turkey)
82. Vázquez M, Rubio F, Houzeaux G, González J, Giménez J, Beltran V, de la Cruz R, Folch A (2014) Xeon phi performance for hpc-based computational mechanics codes
83. Waisman H, Berger-Vergiat L (2013) An adaptive domain decomposition preconditioner for crack propagation problems modeled by XFEM. Int J Multiscale Comput Eng 11(6):633–654
84. Wall WA, Ramm E. Fluid–structure interaction based upon stabilized (ale) finite element method. In: IV World congress on computational mechanics. Barcelona. CIMNE.
85. Wieners C (2001) The application of multigrid methods to plasticity at finite strains. ZAMM J Appl Math Mech [Zeitschrift fr Angewandte Mathematik und Mechanik] 81(S3):733–734
86. Zienkiewicz O, Taylor R (2000) The Finite Elements method for Solid and Structural Mechanics, 5th edn. Butterworth-Heinermann, Boston